Accepted for publication in Theory and Practice of Logic Programming

HTN Planning in Answer Set Programming

JÜRGEN DIX
The University of Manchester, Dept. of Computer Science,
Oxford Road, Manchester, M13 9PL, UK,
email: jdix@cs.man.ac.uk

Ugur Kuter and Dana Nau
University of Maryland, Dept. of Computer Science
College Park, MD 20752, USA
email: ukuter,nau@cs.umd.edu

Abstract

In this paper we introduce a formalism for solving *Hierarchical Task Network* (HTN) Planning using *Answer Set Programming* (ASP). The ASP paradigm evolved out of the stable semantics for logic programs in recent years and is strongly related to nonmonotonic logics. We consider the formulation of HTN planning as described in the *SHOP* planning system and define a systematic translation method from *SHOP*'s representation of the planning problem into logic programs with negation. We show that our translation is *sound* and *complete*: answer sets of the logic program obtained by our translation correspond exactly to the solutions of the planning problem.

Our approach does not rely on a particular system for computing answer sets. It can therefore serve as a means to evaluate ASP systems by using well-established benchmarks from the planning community. We tested our method on various such benchmarks and used *smodels* and *DLV* for computing answer sets.

We compared our method to (1) similar approaches based on non-HTN planning and (2) *SHOP*, a dedicated planning system. We show that our approach outperforms non-HTN methods and that its performance is closer to that of *SHOP*, when we are using ASP systems which allow for nonground programs.

Keywords: HTN-planning, nonmonotonic reasoning, ASP systems, benchmarks

1 Introduction and Related Work

In the past few years, the availability of very fast nonmonotonic systems based on logic programming (LP) made it possible to attack problems from other, non-LP areas, by translating these problems into logic programs and running a fast prover on them. One of the first such system was *smodels* (Niemelä & Simons, 1996) and one of the early applications (Dimopoulos *et al.*, 1997) was to transform planning problems in a suitable way and to run *smodels* on them (see also (Dix *et al.*, 2001)).

Since then more implemented systems with different properties for dealing with logic programs have become available: *DLV* (Eiter *et al.*, 1998), *XSB* (Chen & Warren, 1996; Rao *et al.*, 1997) to cite the most well-known. In addition, the paradigm of Answer Set Programming (ASP) emerged (Apt *et al.*, 1999): the idea is that problems located on the second level of the polynomial hierarchy are well suited to be tackled with the machinery of answer sets. In particular problems which allow for many solutions (like in planning where usually many plans for a given problem exist) fit in this picture.

In this paper, we investigate the ways of formulating and solving HTN planning problems using nonmonotonic logic programs under the ASP semantics. HTN planning (Sacerdoti, 1977; Erol *et al.*, 1994; Wilkins, 1988; Nau *et al.*, 1999) is an AI-planning paradigm in which the goals of the planner are defined in terms of activities (tasks) and the planning process is accomplished by using the techniques of task decomposition. There are several well-known HTN planning systems such as *Universal Method Composition Planner* (*UMCP*) (Erol *et al.*, 1994), *Simple Hierarchical Ordered Planner* (*SHOP*) (Nau *et al.*, 1999), and *SHOP2* (a total-order planner with partially ordered subtasks) (Nau *et al.*, 2001). In this work, we focus on the *SHOP* planning system, which is a domain-independent HTN planning system that is built around the concept called *ordered task decomposition*.

We describe a systematic translation method $\mathfrak{Trans}(\cdot)$ which transforms HTN-planning problems as formalized in *SHOP* into logic programs with negation. Our basic goal is that an appropriate semantics of the logic program should correspond to the solutions (plans) of the planning problem. We have adapted the syntax of the *smodels* software for our transformation, although we are also experimenting with other systems like *DLV* and *XSB*.

1.1 Related Work

There are many efforts in the literature for formulating actions in logic programs and solving planning problems by using formulations such as (Gelfond & Lifschitz., 1998; Turner, 1997; Lifschitz, 1999). (Gelfond & Lifschitz., 1998) describes three different action description languages that formalize theories of actions. The latest one of these languages, the language $\mathcal{C}$, provides means to implement that formalism as logic programs to solve planning problems effectively and efficiently (Lifschitz, 1999; Giunchiglia & Lifschitz, 1998). The $\mathcal{C}$ language consists of general template to define actions that have preconditions and effects. (McCain & Turner, 1997) presents a language for causal theories. They have also developed a system called *Ccalc*, which is a model checker for the language of such causal theories translated from propositions in the $\mathcal{C}$ action language using rewrite rules (?). The idea in all these works is that representing a given computational problem by a logic program whose models correspond to the solutions for the original problem. This idea was the main inspiration for our work presented here.

(Baral & Tuan., 2001) presents a language about actions using causal laws to reason in probabilistic settings and solves the planning problems in such settings. The language resembles similarities to those described above, but the action theory incorporates probabilities and probabilistic reasoning techniques—as described in (Pearl, 1988)—to solve the planning problems with uncertainty.

(Dimopoulos *et al.*, 1997) presents a framework for encoding planning problems in logic programs with negation-as-failure. In this work, the idea is almost the same as ours, that is, the models of the logic program corresponds to the plans. However, this work considers only action-based planning problems and incorporates ideas from such planners *GRAPHPLAN* and *SATPLAN*. In terms of the underlying assumptions and methods presented in (Dimopoulos *et al.*, 1997), our approach is completely different.

(Son *et al.*, 2001) discusses solving planning programs by logic programs. The difference between this work and the one described above is that (Son *et al.*, 2001) incorporates domain-dependent control knowledge to improve the performance of the planning. In this respect, this work is similar to HTN planning algorithms. However, the encoding provided in this work is conceptually not an HTN-planner; instead, it uses hierarchical networks to define domain constraints such as the ordering relationships between the actions, and use them in pruning the search for correct sequence of actions to solve the planning problem.

Our experimental results suggest that both (1) encodings using HTN planning are better than other encodings, because the HTN control knowledge can be used to prune irrelevant branches of the search space; and (2) running an ASP system on non-ground programs (obtained from planning problems) results in a drastic performance relative to *smodels*, thus bringing our method closer to dedicated planning systems like *SHOP*.

1.2 Organization

This paper is organized as follows. We describe in Section 2 the HTN-planning paradigm as well as the *SHOP* planning system. In Section 3 we present our causal theory for HTN-planning and our translation method to transform HTN planning problems into logic programs with negation. Section 4 contains our results. Our main theorem is that our translation method is correct and complete with respect to HTN-planners. We also present our experimental results along with some discussions on the sources of complexity. Finally, we conclude with Section 5 and provide our future research directions.

2 Hierarchical Task Network (HTN) Planning

SHOP is a domain-independent Hierarchical Task Network (HTN) planning algorithm (Nau *et al.*, 1999; Nau *et al.*, 2000). However, one difference between *SHOP* and most other HTN planning algorithms is that *SHOP* plans for tasks in the same order that they will later be executed. Planning for tasks in the order that those tasks will be performed makes it possible to know the current state of the world at each step in the planning process, which makes it possible for *SHOP*'s precondition-evaluation mechanism to incorporate significant inferencing and reasoning power, including the ability to call external programs to reason about preconditions and the ability to perform numeric computations.

In order to do planning in a given planning domain, *SHOP* needs to be given knowledge about that domain. *SHOP*'s knowledge base contains *operators* and *methods*. Each operator is a description of what needs to be done to accomplish some primitive task, and each method is a prescription for how to decompose some complex task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable. More than one method may be applicable to the same task, in which case there will be more than one possible way to decompose that task.

Given the next task to accomplish, *SHOP* chooses an applicable method, instantiates it to decompose the task into subtasks, and then chooses and instantiates other methods to decompose the subtasks even further. If the constraints on the subtasks prevent the plan from being feasible, *SHOP* will backtrack and try other methods.

As an example, Figure 1 shows two methods for the task of travelling from one location to another: *travelling by air*, and *travelling by taxi*. Travelling by air involves the subtasks of purchasing a plane ticket, travelling to the local airport, flying to an airport close to our destination, and travelling from there to our destination. Travelling by taxi involves the subtasks of *calling a taxi*, *riding in it to the final destination*, and *paying the driver*.

Note that each method's preconditions are not used to create subgoals (as would be done in action-based planning). Rather, they are used to determine whether or not the method is applicable: thus in Figure 1, the *travel by air* method is only applicable for long distances, and the *travel by taxi* method is only applicable for short distances.

Now, consider the task of travelling from the University of Maryland to MIT. Since this is a long distance, the *travel by taxi* method is not applicable, so we must choose the *travel by air* method. As shown in Figure 1, this decomposes the task into the following subtasks: *(1)* purchase a ticket from Baltimore-Washington International (BWI) airport to Logan airport, *(2)* travel from the University of Maryland to BWI, *(3)* fly from BWI airport to Logan airport, and *(4)* travel from Logan airport to MIT. For the subtasks of travelling from the University of Maryland to BWI and travelling from Logan to MIT, we can use the *travel by taxi* method to produce additional subtasks as shown in Figure 1.

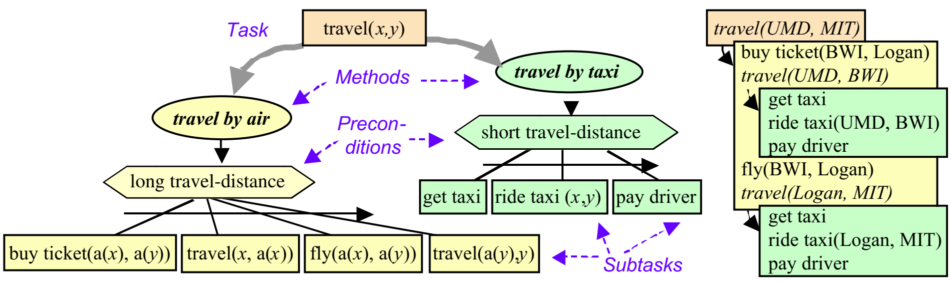

Fig. 1. Travel planning example.

Here are some of the complications that can arise during the planning process:

- The planner may need to recognize and resolve interactions among the subtasks. For example, in planning how to travel to the airport, one needs to make sure one will arrive at the airport in time to catch the plane. To make the example in Figure 1 more realistic, such information would need to be specified as part of *SHOP*'s methods and operators.

- In the example in Figure 1, it was always obvious which method to use. But in general, more than one method may be applicable to a task. If it is not possible to solve the subtasks produced by one method, *SHOP* will backtrack and try another method instead.

SHOP uses the usual first-order logic definitions for atoms, terms, variable and constant symbols, function and predicate symbols, conjuncts, most-general unifiers and Horn clauses. Its domain description consists of methods, operators and axioms as described below.

Definition 1 (Method: (Meth h χ t)) *A* method *is an expression of the form (***Meth** *h χ **t**) where h (the method's* head*) is a compound task, χ (the method's* preconditions*) is a conjunct and **t** is a totally ordered list of subtasks, called the* task list.

Definition 2 (Operator: (Op h χ_{del} χ_{add})) *An* operator *is an expression of the form (***Op** *h χ_{del} χ_{add}), where h (the* head*) is a primitive task and χ_{add} and χ_{del} are lists of atoms (called the* add- *and* delete-lists*). The set of variables in the atoms in χ_{add} and χ_{del} is a subset of the set of variables in h.*

Definition 3 (Axioms: $\mathcal{AX}$) *An* axiom *is an expression of the form*

$$a \leftarrow l_1, \ldots, l_n,$$

where a is an atom and the l_i are literals.

A plan, P, is defined as the sequence of ground operator instances.

Definition 4 (Plans) *A* plan *is a list of heads of ground operator instances. If $P = (p_1 p_2 \ldots p_n)$ is a plan and $\mathcal{S}$ is a state (a set of ground atoms a), then the* result *of applying P to $\mathcal{S}$ is the state result$(\mathcal{S}, P)$ = result(result(... (result$(\mathcal{S}, p_1), p_2), \ldots), p_n)$. A plan P is called a* simple plan *when $n = 1$.*

Definition 5 (Simple reductions) *Let t be a task, $\mathcal{S}$ be the initial state, Meth = (***Meth** *h χ **t**) be a method, and $\mathcal{AX}$ be an axiom set. Suppose that u is a unifier for h and t, and that v is a unifier that unifies χ^u with respect to $\mathcal{S} \cup \mathcal{AX}$. Then the method instance $(Meth^u)^v$ is* applicable *to t in $\mathcal{S}$, and the result of applying it to t is the task list $\mathbf{r} = (\mathbf{t}^u)^v$. The task list **r** is a* simple reduction *of **t** by Meth in $\mathcal{S}$.*

Definition 6 (Domains and problems)
A domain representation *is a set of axioms, operators and methods. A* planning problem *is a triple $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is a state, $\mathbf{t} = (t_1 t_2 \ldots t_k)$ is a task list, and*

$\mathcal{D}$ is a domain representation. Suppose $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is a planning problem and $P = (p_1 p_2 \ldots p_n)$ is a plan. Then we say that P solves $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, or equivalently, that P achieves *$\mathbf{t}$ from $\mathcal{S}$ in $\mathcal{D}$ (we will omit the phrase "in $\mathcal{D}$" if the identity of $\mathcal{D}$ is obvious) if any of the following is true:*

1. *Case 1: $\mathbf{t}$ and P are both empty, (i.e., $k = 0$ and $n = 0$);*
2. *Case 2: t_1 is a primitive task, p_1 is a simple plan for t_1, $(p_2 \ldots p_n)$ achieves $(t_2 \ldots t_k)$ from result$(\mathcal{S}, p_1)$;*
3. *Case 3: t_1 is a composite task, and there is a simple reduction $(r_1 \ldots r_j)$ of t_1 in $\mathcal{S}$ such that P achieves $(r_1 \ldots r_j t_2 \ldots t_k)$ from $\mathcal{S}$.*

The planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is solvable *if there is a plan that solves it. We therefore denote the set of all plans by $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$.*

3 Encoding HTN planning in Nonmonotonic Logic Programming

Our approach of encoding HTN-planning problems as logic programs is based on *SHOP*'s representation of a planning problem. We first describe *SHOP*'s formalism for HTN-planning briefly. Then we present first steps of a causal theory of HTN planning based on that formalism. This theory serves as a motivation for our translation methodology which is given in the subsequent subsection. We conclude this section with the formalization of a particular example.

3.1 Formal Definitions for HTN-planning: Syntax and Semantics

We use the same definitions for variable and constant symbols, predicate symbols, terms, atoms as *SHOP*. Our definitions for axioms, operators, methods are adapted from *SHOP*. The next paragraph describes these concepts briefly; for a detailed discussion see (Nau *et al.*, 1999).

A *term* is either a constant or a variable symbol. A *state* $\mathcal{S}$ is a set of ground atoms, and an *axiom* is a Horn clause. A task is an expression of the form $(h t_1 t_2 \ldots t_n)$, where h (the task's name) is a task symbol, and $t_1, t_2, \ldots, t_n$ (the task's arguments) are terms. A task can be either primitive or composite. A task list is a list of tasks.

An operator specifies a primitive task that can be accomplished by modifying the current state of the world by removing every atom in its deletions list and adding every atom in its additions list. As an example, here is a possible implementation of the `get-taxi` operator from Figure 1:

```
(:Op(!get-taxi ?x)
   ((service-available-to ?x))
   ((taxi-coming-to ?x)))
```

Here is a possible implementation of the `travel-by-taxi` method from the same figure:

```
(:Meth (travel ?x ?y)
  ((smaller-distance ?x ?y))
  ((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
```

3.2 Causal Theory for HTN-planning

In this section we prepare the ground for our translation in the next subsection. We give some definitions of a causal theory for HTN-planning in a *SHOP*-like ordered task decomposition. The reason for presenting this causal theory is not to give a formal semantics, but to give some motivations for the more technical aspects of the translation given later on.

In the definitions below, $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is a planning problem as introduced in Definition 6.

Definition 7 (Caused) *Let $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be a planning problem and let P be a plan.*

We define for a ground literal, l, the property of being caused *wrt. $\mathcal{S}$. This property is defined through the following recursive definition:*

1. l caused $\mathsf{wrt.}$ $\mathcal{S}$ *if* $\begin{cases} a \in \mathcal{S} & \textit{if } l = a, \\ a \notin \mathcal{S} & \textit{if } l = \neg a. \end{cases}$
2. l caused $\mathsf{wrt.}$ $\mathcal{S}$*: if there is an axiom given in the domain description $\mathcal{D}$ of the form*
$$a \leftarrow l_1 \wedge l_2 \wedge \ldots \wedge l_n,$$
such that $l = a$ and every l_i is caused wrt. $\mathcal{S}$: l_i caused $\mathsf{wrt.}$ $\mathcal{S}$.

A list of literals, L, is caused $\mathsf{wrt.}$ $\mathcal{S}$ *iff every literal in L is* caused $\mathsf{wrt.}$ $\mathcal{S}$.

The next definition represents an important persistence property over time.

Definition 8 (Law of Inertia) *A ground literal l, which is caused in the current state $\mathcal{S}$, is also caused in the next state $\mathcal{S}'$ unless the negated literal $\neg l$ is caused in $\mathcal{S}'$. Here the symbol $\neg$ denotes classical negation. The Law of inertia can be represented by the following rule:*

$$l \ \mathsf{caused\ \ wrt.}\ \mathcal{S}' : \quad \begin{array}{l} \textit{if } l \ \mathsf{caused\ \ wrt.}\ \mathcal{S} \textit{ and} \\ \textit{not ``}\neg l \ \mathsf{caused\ \ wrt.}\ \mathcal{S}'\textit{''.} \end{array}$$

This rule ensures that for each atom a and each state $\mathcal{S}$, either a or $\neg a$ is caused $\mathsf{wrt.}$ $\mathcal{S}$.

Definition 9 (Caused Tasks) *A primitive task t is* caused *(to-be-accomplished)* $\mathsf{wrt.}$ *$(\mathcal{S}, \mathcal{D})$ iff there exists an operator for t: (**Op** t χ_{del} χ_{add}) $\in \mathcal{D}$.*

A composite task t is caused $\mathsf{wrt.}$ *$(\mathcal{S}, \mathcal{D})$ iff*

1. *there exists a method for t: (**Meth** t χ $\mathbf{t}$) $\in \mathcal{D}$,*
2. *the preconditions-list χ, which is a list of literals representing a conjunct, is caused, and*
3. *all of the successor subtasks of $\mathbf{t}$ are caused. In that case, we say the subtasks cause t.*

Using this causal theory as an intermediate step, we developed a systematic translation method for mapping planning problems to logic programs with negation which we illustrate in the next section.

Theorem 10
Let a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be given, where $\mathcal{S}$ is the initial state, $\mathbf{t}$ is the list of tasks to be achieved and $\mathcal{D}$ is the domain description.

If there is a solution to $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, then each of the tasks in $\mathbf{t}$ is caused wrt ($\mathcal{S}$, $\mathcal{D}$) in the order they are given in $\mathbf{t}$.

Proof

The proof starts by recursively defining the solution of an HTN-planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ and showing the causal relationships based on our causal theory at the same time.

The solution plan, $P((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, for the planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ is initially empty. If $\mathbf{t}$ is empty, then $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ contains exactly one plan, namely the empty plan. This is because of the fact that there will be no tasks to be accomplished—thus, no task to be caused . If $\mathbf{t}$ is not empty, then let h be the first task in $\mathbf{t}$, and let $\mathbf{R}$ be the remainning tasks. There are two cases.

1. If h is primitive and there is no simple plan in $\mathcal{D}$ for it, then $P((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ is empty: there is no solution.
2. If h is primitive and there is a simple plan p in $\mathcal{D}$ for $\mathbf{t}$, then $P((\mathcal{S}, \mathbf{t}, \mathcal{D})) = \mathbf{append}(p, q)$, where $q \in \mathbf{Sol}(result(\mathcal{S}, p), \mathbf{R}, \mathcal{D})$. Then, according to the first part of Definition 9 of our causal theory, we say that h is caused wrt. $\mathcal{D}$.
3. If h is a composite task, then $P((\mathcal{S}, \mathbf{t}, \mathcal{D})) = P(S, append(r, R), D)$, where r is one of the simple reductions of h (see Definition 5), which is a list of subtasks of h. In order for h to be accomplished—so that $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ will be solvable—all of the subtasks in r have to be accomplished. According to the second part of our Definition 9, this corresponds the fact that in order for h to be caused wrt. $\mathcal{D}$, all of its subtasks must be caused wrt. $\mathcal{D}$.

Therefore, it follows from the recursive construction above that if task h is accomplished according to our causal theory, it must be caused as well. If we have more than one task in $\mathbf{t}$, then according to Definition 6, we have to accomplish all of them separately in the order they are given in $\mathbf{t}$, which also means that each of them must be caused wrt. ($\mathcal{S}$, $\mathcal{D}$) in that particular order. $\square$

3.3 Encoding Planning Problems as Logic Programs

Translating a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ to its logic program counterpart $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ requires encoding the methods, the operators, and the axioms as logic program segments as well as the underlying ordered task decomposition characteristics of *SHOP*. For this reason, we present our translation method in several steps, performing all of which yield a logic program that is capable of solving planning problems in the way *SHOP* does.

Step 1. Encoding the Domain Independent Rules.

The problem independent rules for a logic program are adapted mainly from (Son *et al.*, 2001; Lifschitz, 1999; Dimopoulos *et al.*, 1997). The main atoms in these rules are

- $state(A, T)$: A holds in the current state at time T,
- $literal(A)$: A is a literal,
- $contrary(A, \neg A)$: A and its negation $\neg A$ are contradictory.

The main rules are given next. In these rules, T is a variable of the sort time.

$$
\begin{array}{lll}
literal(A) & :- & atom(A). \\
literal(neg(A)) & :- & atom(A). \\
contrary(A, neg(A)) & :- & atom(A). \\
contrary(neg(A), A) & :- & atom(A). \\
state(A, T+1) & :- & literal(A), literal(B), contrary(A, B), state(A, T), \\
 & & not\, state(B, T+1).
\end{array}
$$

Here, the first and the second rules encode the fact that any atom and its negation is a literal, and the last rule is the Law of Inertia.

Step 2. Encoding the Problem Dependent State Elements.

SHOP allows using variables in the domain descriptions of the planning problems. Unfortunately most nonmonotonic systems can not handle free variables. For example *smodels* is doing an (intelligent) grounding of the description it is given and it is requiring that a certain syntactic condition, safeness, is satisfied. *DLV* allows variables, but imposes a safeness restriction and does not allow function symbols. In the current implementation of *smodels*, there is a further technical condition (which will be relaxed in the next release) that we have to take into account: For every variable that we use in our logic program, we have to specify the range of values that it can be instantiated during model generation process.

Due to this fact, we have to take care of the following:

1. we have to enumerate all the possible ground atoms that can be used in the logic program;
2. we have to include rules for type predicates in the logic program, which define the range of values for a variable of a certain type.

Translation Procedure for the Atoms and Type predicates

1. Specify the atoms that can ever be used by the logic program as $atom(\_)$.
2. Define a type of the form $[type](\_)$ for each variable that is used in the logic program, and specify the range of values for each of those types.

To give an example, consider the method for travelling from one place to another. Suppose that the name of this method is $travel(X, Y)$. Here, X and Y are variables of locations that are going to be used in this method. The logic program that encodes this method must have the following type predicates: $place(umd)$, $place(mit)$, and so on, for all locations that can be used in the model generation process.

Note that if the translation is being done for a system that cannot handle free variables - like *smodels*-, we have to specify the type of each variable appearing in each rule of the translated logic program by adding the necessary type predicates to the righthand side of those rules. On the other hand, in a system like DLV, which can handle free variables, we may omit the type predicates in the rules as long as we do not violate the safeness restrictions.

Step 3. Encoding the Initial State for the Planning Problems.

SHOP's initial state for the planning problems is defined to be a set of ground atoms. In this respect, given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the following procedure must be used in order to translate it into its logic program counterpart.

Definition 11 ($\mathfrak{Trans}(\mathcal{S})$: Translation for Initial State)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for all $a \in \mathcal{S}$, add the rule

$$state(a, 0) :-$$

This rules specifies that a is in the state at time 0, which is used to designate the initial time.

Step 4. Encoding the Goal Task(s).

The goal tasks are the ordered list of tasks that must be accomplished by the planning system. In our translation, this list is encoded in the following rules. In these rules, T_i's are variables of time, and h_i's are the names of the tasks that must be accomplished, and $\mathsf{Pre}(h_i)$'s are the labels for the precondition lists of the methods which were applied to those tasks.

Definition 12 ($\mathfrak{Trans}(\{h_1, \ldots, h_n\})$: Translation for Goal Tasks)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, let $\mathbf{t} = h_1, h_2 \ldots, h_n$ be the ordered sequence of goal tasks. Then,

1. *Encode the first goal task h_1 as the following rule:*

$$currentTask(h_1, 0) \ :-$$

2. *For the rest of the goal tasks, add the following $n-1$ rules ($i = 2, \ldots, n$) to the logic program:*

$$currentTask(h_i, T_i) \ :- \ caused(h_{i-1}, \mathsf{Pre}(h_{i-1}), T_{i-1}, T_i).$$

The predicate $currentTask(task\_name, T)$ uniquely specifies the current task selected at time T. As described above, we assumed that the planning process begins at time 0. If there exists only one goal task to be accomplished for the problem in hand, then only defining the first rule will suffice.

Definition 12 enforces the fact that a goal task h_i is designated as the current task to be accomplished if the previous goal task h_{i-1} in $\mathbf{t}$ is caused. This is a direct consequence of our Theorem 10. Following this definition and theorem, we add the following rules in the logic program:

$$plan\_found \ :- \ caused(h_n, \mathsf{Pre}(h_n), T_n, T_{n+1}).$$

$$:- \ not\, plan\_found.$$

where T_n denotes the time when the particular method for the last goal task, h_n, is decomposed and T_{n+1} is the time at which h_n is caused (accomplished).

These two rules together state that if the last goal task is caused then there is a plan (solution) for the planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$ as a result of Definition 12. Otherwise, there is none.

Step 5. Encoding the Problem Dependent Control Structures.

Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the domain description $\mathcal{D}$ contains axioms, operators and methods as described in the previous section. For each of these constructs, we present a translation procedure.

Definition 13 (Characteristic Function for Literals)
Given a literal, l, we define $C(l,T)$, the characteristic function of l at time T, as

$$C(l,T) := \begin{cases} state(a,T) & if\ l = a, \\ not\ \ state(a,T) & if\ l = \neg a. \end{cases}$$

where a is an atom.

Definition 14 ($\mathfrak{Trans}(\mathcal{AX})$: Translation for Axioms)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for all "$a \leftarrow l_1, \ldots, l_n$" $\in \mathcal{D}$, add the rule

$$state(a,T) \ :- \ C(l_1,T), C(l_2,T), \ldots, C(l_n,T),$$

where $C(l_i,T)$ is defined in Definition 13 above.

Definition 15 ($\mathfrak{Trans}(\mathcal{OP})$: Translation for Operators)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, for all $\mathsf{Op} \in \mathcal{OP}$, add the following rules:

for all $a \in \mathrm{Del}(\mathsf{Op})$:

$$state(neg(a), T+1) \ :- \ currentTask(h,T).$$

and for all $a \in \mathrm{Add}(\mathsf{Op})$:

$$state(a, T+1) \ :- \ currentTask(h,T).$$

and finally add the following rules,

$$action(h, T, T+1) \ :- \ currentTask(h,T).$$

$$caused(h, T, T+1) \ :- \ currentTask(h,T).$$

The first and the second rule encode the delete- and the add-lists of the operator respectively. The third rule designates the action for to accomplish the primitive task h. Since, in *SHOP*, the ground instances of operators represent action, we do not need such designations there. The last rule encodes the fact that the task is caused if and only if there exists an operator for it (see Definition 9).

As described in the previous section, given a composite task h, a method m can be categorized as one of the following two types: either m can be the only method for the task h or it can be one of the many methods for task h. Although they possess slight differences, the translation procedures for each of these cases are mostly similar. In the rest of this section, we will present these translation procedures.

Definition 16 ($\mathfrak{Trans}(\mathcal{METH})$: Translation for Methods)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, we are translating the methods contained in $\mathcal{D}$.

Case 1: *Given a composite task h, suppose that there exists only one method m whose head matches with h. Let $\mathsf{Pre}(h)$ be the label for the precondition list of the method m and let Z be the set of all variables that are used in that precondition list. Then,*

1. *Designate the method to be applied to the current composite task:*

$$method(h, \mathsf{Pre}(h), T) \ :- \ currentTask(h, T).$$

2. *Define the precondition list of the method by inserting the following rule(s) in the logic program (where $l_1, \ldots, l_n$ are all preconditions of the method m): for $i = 1, \ldots, n$,*

   (a) *If l_i is a positive literal and it contains free variables (i.e, it contains variable symbols that do not appear in the head of the method m). Let $a_i(X_1, X_2, \ldots, X_v)$ denote the fact that there are v free variables in l_i. Let s_j denote the number of substitutions for the variable X_j in the current state $\mathcal{S}$. For each $k_j = 1 \ldots s_j$, define $Y_m = (X_{1,k_1}, X_{2,k_2}, \ldots, X_{n,k_n})$, where $m = 1, \ldots, s_1 \times s_2 \times \ldots \times s_v$ such that $Y_1 = (X_{1,1}, X_{2,1}, \ldots, X_{v,1})$ and $Y_m = (X_{1,s_1}, X_{2,s_2}, \ldots, X_{n,s_v})$.*
   Then, add the following rule in the logic program,

$$\begin{aligned} checked(state(a_i(Y_1), T), T) \ :- \ & method(h, \mathsf{Pre}(h), T), \\ & state(a_i(Y_1), T), \\ & \textstyle\bigwedge_{m=2}^{s_1 \times \cdots \times s_v} not\, checked(state(a_i(Y_m), T), T), \\ & \textstyle\bigwedge_{j=1}^{v} X_{j,1}! = X_{j,2}! = \ldots ! = X_{j,s_j}. \end{aligned}$$

   (b) *Otherwise, add the following rule,*

$$checked(state(l_i, T), T) \ :- \ C(l_i, T), method(h, \mathsf{Pre}(h), T).$$

   where $C(l_i, T)$ is as defined in Definition 13.

   and finally, add the following rule in the logic program,

$$\begin{aligned} preCond(\mathsf{Pre}(h), Z, T) \ :- \ & checked(state(l_1, T), T), \ldots, checked(state(l_n, T), T), \\ & method(h, \mathsf{Pre}(h), T). \end{aligned}$$

3. *Assuming the ordered task decomposition for this method is the ordered set of tasks $\{t_1, t_2, , t_n\}$, add the following set of rules to the logic program to specify the decomposition (note that the time variable T_1 in the following rule definitions in this item denote the same value as the time variable T in the rule definitions presented in other items does):*

$$
\begin{array}{lcl}
currentTask(t_1, T_1) & :- & method(h, \mathtt{Pre}(h), T_1), preCond(\mathtt{Pre}(h), Z, T_1). \\
currentTask(t_2, T_2) & :- & method(h, \mathtt{Pre}(h), T_1), \\
& & preCond(\mathtt{Pre}(h), Z, T_1), \\
& & caused(t_1, \mathtt{Pre}(t_1), T_1, T_2), \\
& & T_2 \geqq T_1. \\
\vdots & \vdots & \qquad\qquad\qquad \vdots \\
currentTask(t_n, T_n) & :- & method(h, \mathtt{Pre}(h), T_1) \\
& & preCond(\mathtt{Pre}(h), Z, T_1), \\
& & \bigwedge_{i=1}^{n-1} caused(t_i, \mathtt{Pre}(t_i), T_i, T_{i+1}), \\
& & \bigwedge_{i=2}^{n-1} T_{i+1} \geqq T_i.
\end{array}
$$

4. *Finally, specify the causal links from the child tasks $\{t_1, t_2, , t_n\}$ to the parent task h.*

$$
\begin{array}{lcl}
caused(h, \mathtt{Pre}(h), T, T_{n+1}) & :- & method(h, \mathtt{Pre}(h), T), \\
& & preCond(\mathtt{Pre}(h), Z, T), \\
& & \bigwedge_{i=1}^{n} caused(t_i, \mathtt{Pre}(t_i), T_i, T_{i+1}), \\
& & \bigwedge_{i=2}^{n+1} T_i \geqq T_{i-1}.
\end{array}
$$

Case 2: *Given a composite task h, suppose that there exist $n > 1$ many methods m_i such that $i = 1, \ldots, n$, whose heads match with h. Then for each such method m_i, perform the previous procedure given for Case 1 above, with the following modifications:*

1. *Replace each* $\mathtt{Pre}(h)$ *term with the term* $\mathtt{Pre}(h)_i$.
2. *Use* $PreCond(m_i)$ *for the particular* m_i.
3. *Rewrite the first rule as follows:*

$$
\begin{array}{lcl}
method(h, \mathtt{Pre}(h)_i, T) & :- & currentTask(h, T), \\
& & \bigwedge_{k=1,\ldots,n, k\neq i} not\ method(h, \mathtt{Pre}(h)_k, T)
\end{array}
$$

The rest of the rules are exactly the same as presented for the Case 1.

3.4 A Translation Example: An Elevator Domain

In this section, we give an example translation for the Miconic-10 Elevator domain, which was introduced as an official benchmark domain during the AIPS-2000 competition (see (Bacchus, 2001) and `http://www.cs.toronto.edu/aips2000`). In the competition, the domain was configured in a number of versions to accommodate the representational power of different planning systems. Its simplest version (the one referred to as the "first track" version at `http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html`) was one of the test cases in (Son *et al.*, 2001), and we used the same version here and in our experiments (see Section 4.2). This version differs from the more complicated versions in the following respects: The planner simply has to generate plans to serve a group of passengers of whom the origin and destination floors are given. There are no constraints such as satisfying space requirements of passengers or achieving optimal elevator controls.

Although we have the full translation of the domain, for the sake of simplicity, we give here only a part of our translation along with the corresponding HTN domain description for comparison.

Suppose that we have only two floors and one person to be delivered. Furthermore, suppose that the elevator starts its operation at the 0^{th} floor (i.e., the ground floor which is marked as 0) and its initial direction is upwards. Our passenger boards the elevator at the first floor and wants to go down to the ground floor.

Now, we will describe the basics of the translation process step by step as described in the previous section.

Step 1. Encoding the domain independent rules.

For our elevator example, the domain independent rules are as follows:

$$\begin{array}{lll}
literal(P) & :- & atom(P). \\
literal(neg(P)) & :- & atom(P). \\
contrary(P, neg(P)) & :- & atom(P). \\
contrary(neg(P), P) & :- & atom(P). \\
state(P, T+1) & :- & time(T), literal(P), literal(Q), contrary(P, Q), \\
 & & state(P, T),\ not\ state(Q, T+1).
\end{array}$$

As it can be seen in these rule definitions, we have to define all possible atoms that can ever be used during planning.

Step 2. Encoding the Problem Dependent State Elements.

In this step, we have to formulate all of the possible atoms that can ever be used during the planning process. Due to the fact that the variables in *smodels* semantics must have a range of values, we must also define type predicates in our translation as described in the previous section. In a *SHOP* domain description, we do not need to make these definitions about the set of all possible atoms and tasks, nor about the type predicates.

In our elevator example, we need the following rules for defining the set of all possible atoms:

$$
\begin{array}{lll}
atom(boarded(P)) & :- & person(P). \\
atom(goal(P)) & :- & person(P). \\
atom(served(P)) & :- & person(P). \\
atom(lift(F)) & :- & floor(F). \\
atom(origin(P,F)) & :- & person(P), floor(F). \\
atom(destination(P,F)) & :- & person(P), floor(F). \\
atom(top(F)) & :- & floor(F). \\
atom(bottom(F)) & :- & floor(F). \\
atom(current\_direction(X)) & :- & direction\_type(X). \\
 & \vdots &
\end{array}
$$

And also the type predicates such as:

$$
\begin{array}{ll}
person(0) & :- \\
floor(0\ldots1) & :- \\
direction\_type(up) & :- \\
direction\_type(down) & :- \\
 & \vdots
\end{array}
$$

Step 3. Encoding the Initial State for the Planning Problems.

We need the following rules to specify the initial state in our encoding of the elevator example:

$$
\begin{array}{ll}
state(lift(0),0) & :- \\
state(goal(0),0) & :- \\
state(origin(0,1),0) & :- \\
state(destination(0,0),0) & :- \\
state(current\_direction(up),0) & :- \\
 & \vdots
\end{array}
$$

As it can be seen from these rules, these rules specify certain ground atoms to be in the state of the planner (Definition 11 in the previous section). The last argument for each $state(P,T)$ predicate is the time T at which the atom P holds. We define the initial time for the execution of the planner to be 0.

Step 4. Encoding the Goal Task(s).

We need the following rule to specify the goal tasks in our encoding of the elevator problem in which we have only one in this case:

$$currentTask(serve\_all, 0).$$

This rule specifies that our goal task is a task whose head is *serve_all* and the initial time is 0.

Since there is only one goal task in this particular elevator problem, we need only one rule for specifying it. If we had more than one goal task, then we would have several rules as given in Definition 11 in the previous section. To give an example, suppose that we have another goal task *initialize_elevator* which guarantees that the elevator is returned to ground level after serving all the passengers. In the ordered task list definition of *SHOP*, these will constitute a goal task list $\mathbf{t} = (serve\_all, initialize\_elevator)$. In this case, we would have the following rules according to Definition 11 in our encoding of the problem:

$$
\begin{array}{lll}
currentTask(serve\_all, 0) & :- & \\
currentTask(initialize\_elev, T) & :- & caused(serve\_all, \mathtt{Pre}(serve\_all), 0, T), \\
& & T > 0.
\end{array}
$$

Step 5. Encoding the Problem Dependent Control Structures.

In this step, we formulate our axioms, operators, and methods, which are given in the HTN domain description, D.

For example, in the HTN definition of our particular elevator example, we have the following operator definition for moving the elevator from one floor to another:

```
(:operator (!move f1 f2)
        (lift(f1))
        (lift(f2)))
```

This operator is encoded (see Definition 15.) in the following set of rules:

$$
\begin{array}{lll}
state(neg(lift(F1)), T+1) & :- & time(T), floor(F1), floor(F2), \\
& & currentTask(move, F1, F2, T). \\
state(lift(F2), T+1) & :- & time(T), floor(F1), floor(F2), \\
& & currentTask(move, F1, F2, T). \\
action(move, F1, F2, T, T+1) & :- & time(T), floor(F1), floor(F2), \\
& & currentTask(move, F1, F2, T). \\
caused(move, F1, F2, T, T+1) & :- & time(T), floor(F1), floor(F2), \\
& & currentTask(move, F1, F2, T).
\end{array}
$$

These rules apply only when the current task is the primitive task $(move f1 f2)$.

There are two cases in the translation of a method as given in Definition 16. In the first case, we may have only one HTN method for a particular task. Suppose that the current task is $(op\_elev f)$, where f is a floor, and we have the following method:

```
(:method (op_elev f)
        (lift(f))            % the precondition
        ((check_board f)     % subtask 1
         (check_dep f)       % subtask 2
         (move_elev f)))     % subtask 3
```

According to Case 1 of the Definition 16, we have the following rules:

$$
\begin{array}{ll}
method(op\_elev, F, \mathtt{Pre}(op\_elev), T) & :- time(T), floor(F), \\
 & \quad currentTask(op\_elev, F, T). \\
checked(state(lift(F), T), T) & :- state(lift(F), T), \\
 & \quad method(op\_elev, F, \mathtt{Pre}(op\_elev), T). \\
preCond(\mathtt{Pre}(op\_elev), F, T) & :- checked(state(lift(F), T), \\
 & \quad method(op\_elev, F, \mathtt{Pre}(op\_elev), T). \\
currentTask(check\_board, F, T) & :- time(T), floor(F), preCond(\mathtt{Pre}(op\_elev), F, T), \\
 & \quad method(op\_elev, F, \mathtt{Pre}(op\_elev), T). \\
currentTask(check\_dep, F, T2) & :- time(T), time(T2), floor(F), \\
 & \quad preCond(\mathtt{Pre}(op\_elev), F, T), \\
 & \quad method(op\_elev, F, \mathtt{Pre}(op\_elev), T), \\
 & \quad caused(check\_board, F, \mathtt{Pre}(check\_board), T, T2), \\
 & \quad T2 \geq T. \\
currentTask(move\_elev, F, T3) & :- time(T), time(T2), time(T3), floor(F), \\
 & \quad preCond(\mathtt{Pre}(op\_elev), F, T), \\
 & \quad method(op\_elev, F, \mathtt{Pre}(op\_elev), T), \\
 & \quad caused(check\_board, F, \mathtt{Pre}(check\_board), T, T2), \\
 & \quad caused(check\_dep, F, \mathtt{Pre}(check\_dep), T2, T3), \\
 & \quad T2 \geq T, T3 \geq T2. \\
caused(op\_elev, F, \mathtt{Pre}(op\_elev), T, T4) & :- time(T), time(T2), time(T3), time(T4), \\
 & \quad floor(F), method(op\_elev, F, \mathtt{Pre}(op\_elev), T), \\
 & \quad preCond(\mathtt{Pre}(op\_elev), F, T), \\
 & \quad caused(check\_board, F, \mathtt{Pre}(check\_board), T, T2), \\
 & \quad caused(check\_dep, F, \mathtt{Pre}(check\_dep), T2, T3), \\
 & \quad caused(move\_elev, F, \mathtt{Pre}(move\_elev), T3, T4), \\
 & \quad T2 \geq T, T3 \geq T2, T4 \geq T3.
\end{array}
$$

In this translation, the first rule designates the application of the method whose head is *op_elev* to the current task $(op\_elev f)$. The second and the third rules make sure that all of the preconditions of the method are satisfied in the current state $\mathcal{S}$. The fourth through the sixth rules define the successor subtasks with the order they were defined in the corresponding HTN method. Note that, in HTN formalism, the ordering of the subtasks enforce the fact that a subtask t can be selected as the current task for decomposition only if all of the subtasks preceding t are accomplished successfully. This is achieved in our translation by the caused properties of the tasks (see Definition 9).

As the other case, we may have two different methods for the same composite task in our HTN domain description. For example, suppose that we also have the following method for the task $(op\_elev f)$ in addition to that given above:

```
(:method (op_elev f)
      (lift(f))          % the precondition
      ((move_elev f)))   % the subtask
```

In a *SHOP*-like HTN planning algorithm, having two methods with different successor subtasks applicable for a particular task creates a branching (backtracking) point in the search space of the planner. If we require the planner to return all the solutions (plans) for the planning problem at hand, the planner should try each branch to find a possibly—but not necessarily—different plan. To be able to implement this property in our translation, we use the NAF literals to generate different answer sets corresponding to the application of each method—as given in Case 2 of Definition 16. According to this definition, we will the following method designation rules in the translation of each method:

$$\begin{array}{lcl} \mathit{method}(\mathit{op\_elev}, F, \mathtt{Pre}(\mathit{op\_elev})_1, T) & :- & \mathit{time}(T), \mathit{floor}(F), \\ & & \mathit{currentTask}(\mathit{op\_elev}, F, T), \\ & & \mathit{not}\, \mathit{method}(\mathit{op\_elev}, F, \mathtt{Pre}(\mathit{op\_elev})_2, T) \\ \mathit{method}(\mathit{op\_elev}, F, \mathtt{Pre}(\mathit{op\_elev})_2, T) & :- & \mathit{time}(T), \mathit{floor}(F), \\ & & \mathit{currentTask}(\mathit{op\_elev}, F, T), \\ & & \mathit{not}\, \mathit{method}(\mathit{op\_elev}, F, \mathtt{Pre}(\mathit{op\_elev})_1, T) \end{array}$$

The index of the predicate $\mathtt{Pre}(h)$, where h is the head of the method - which is *op_elev* in this example-, specifies which branch is being taken. The rest of the translation for each method remains the same as in the example above except all the instances of $\mathtt{Pre}(h)$ should be replaced by $\mathtt{Pre}(h)_i$, where i is the index of the method.

4 Results: Theory and Practice

In this section, we present our theoretical results on the correctness of our translation method and the soundness and the completeness of the resulting logic programs as planning systems as well as the experiments we have undertaken.

4.1 Soundness and Completeness

Due to space limitations, we will not present the whole proofs here, but we will discuss the basic ideas behind them.

Our first theorem states that our translation indeed corresponds to HTN planning as done in *SHOP*.

Let $\mathfrak{Trans}(\cdot)$ be the translation method described in the previous section. Given any HTN-planning problem, we are interested in the relationship between the models (or answer sets) of $\mathfrak{Trans}(\cdot)$.

Theorem 17 ($\mathfrak{Trans}(\cdot)$ and HTN planning) *Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, where $\mathcal{S}$ is the initial state, $\mathbf{t}$ is the list of tasks to be achieved and $\mathcal{D}$ is the domain description, let $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ be the corresponding logic program with negation. Furthermore, let $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ be the set of solutions returned by SHOP. Then,*

1. *If $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D}) = \emptyset$, then $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ has no answer sets.*

2. *If* $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D}) \neq \emptyset$*, then for every plan* $P \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D}) \neq \emptyset$*, there is an answer set of* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$*, such that the action(_) predicates correspond exactly to the steps* p_i *in* P*.*

Proof

Sketch Given an HTN planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the proof starts with defining the solution depth of a plan $P \in \mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ to be the number of decompositions needed to produce P from $\mathbf{t}$ plus the length of P. Then, it follows from the causal theory for HTN-planning described in the previous section (Theorem 10), that both *SHOP* and the logic program $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ have the same set of plans initially. Then, by induction on the solution depth of any solution in $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$, we show that the theorem holds throughout the plan/model generation process. This is because the logic program $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ is produced by our translation methodology, which is based on our causal theory. □

Soundness and completeness are the two important requirements for any planning system. Soundness means that all of the plans that are generated by the planner are actually true solutions to the given planning problem; that is, no plan, which is not solution to the problem, should be generated. Completeness means that the planning system must be able to generate all of the possible plans (solutions) for the given problem.

Corollary 18 (Soundness and Completeness of ASP using $\mathfrak{Trans}(\cdot)$)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$*, where* $\mathcal{S}$ *is the initial state,* $\mathbf{t}$ *is the list of tasks to be achieved and* $\mathcal{D}$ *is the domain description, let* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *be the corresponding logic program with negation. Furthermore, let* $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ *be the set of solutions returned by SHOP.*

Then, the answer sets of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *correspond exactly to the plans in* **Sol** $(\mathcal{S},\mathbf{t},\mathcal{D})$*. There is a bijection between these two sets and each plan in* **Sol** $(\mathcal{S},\mathbf{t},\mathcal{D})$ *can be reconstructed from its corresponding answer set in* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *and vice versa.*

The corollary follows easily from the theorem and the fact that *SHOP* itself has been shown to be a sound and complete planner.

Definition 19 (Solution Tree for $\mathfrak{Trans}(\cdot)$)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$*, and the corresponding logic program with negation* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$*, we define the solution tree produced by* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$*. It is an AND-OR tree,* $\mathcal{T}$*, in which the AND branches represent the task decompositions and the OR branches represent different possible methods whose heads match with a particular task.*

We say $\mathcal{T}$ represents $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$. Without loss of generality, we assume that the solution tree of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ is a complete AND-OR tree as shown in Figure 2. Furthermore, we suppose that $\mathbf{t}$ contains only one task to be accomplished and we have no negated atoms in the preconditions of the methods in $\mathcal{D}$.

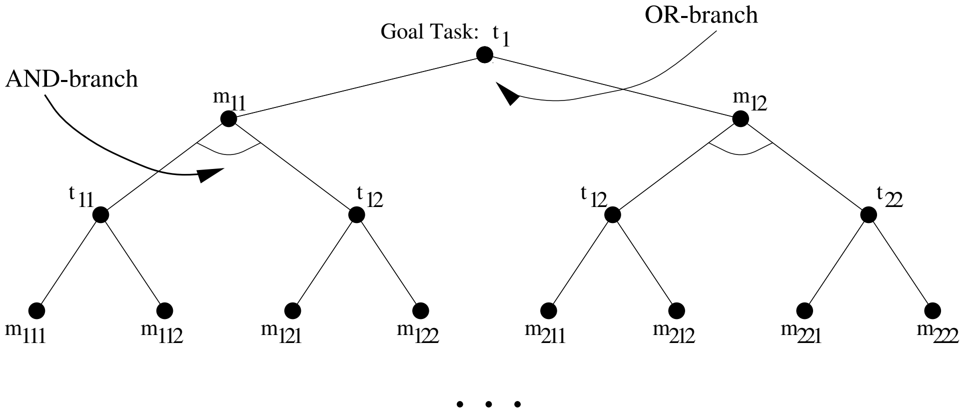

Fig. 2. A complete AND-OR Tree

Theorem 20 (Time Performance using $\mathfrak{Trans}(\cdot)$ (1))
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the corresponding logic program with negation $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, and the solution tree $\mathcal{T}$, then, the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate the set of all answer sets that correspond to $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ monotonically increases with the number of applicable methods for a particular task (the number of branches in a particular OR-branch).

Proof
Sketch Given a particular task t in the solution tree $\mathcal{T}$, let n denote the number of applicable methods to t. According to our translation methodology, for each such method, $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ will include the rules given in Definition 16. Without loss of generality, we assume that $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ requires a unit amount of time to make a rule ground and fire it, and all of the possible method applications to t lead to isomorphic sub-trees. Then, let the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to solve a sub-tree of $\mathcal{T}$ whose root is t, be denoted by $c \neq 0$. The overall time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to find all answer sets is $nc + a$, where a represents the time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ for the rules corresponding to the rest of the parts of the solution tree $\mathcal{T}$.

The proof starts by showing that when $n = 0$, the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to find all answer sets is equal to a and it monotonically increases if we increase n by 1 ($c + a$ is clearly greater than a). Suppose that for all $i < n$, the theorem holds. Then, by induction, if we increase the number of methods applicable to task t to be $n + 1$, the overall time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to find all answer sets will be $(n + 1)c + a$ such that $(n + 1)c + a > nc + a$.

It follows therefore that the total time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate all answer sets that correspond to $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ monotonically increases with the increasing number of applicable methods to a particular task in $\mathcal{T}$. $\square$

Theorem 21 (Time Performance using $\mathfrak{Trans}(\cdot)$ (2))
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the corresponding logic program with negation $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, and the solution tree $\mathcal{T}$, then, the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$

to generate the set of all answer sets that correspond to $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ *monotonically increases with the number of subtasks of a particular task (the number of branches in an AND-branch).*

Proof

Given a particular task t and a method m : $(\mathbf{Meth}\ h\ \chi\ \mathbf{t})$, whose head, h, matches with the task t, the subtasks of the task t correspond the simple reduction r of t by in $\mathcal{S}$. Suppose that there are n subtasks in r. Then, according to our translation methodology, $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ will contain n rules as shown in the item 3 of the first case in Definition 16. Without loss of generality, we assume that $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ requires a non-zero unit amount of time, denoted as c, to make a rule ground, and fire it. Let $k \neq 0$ be the number of occurrences of the particular task t in the solution tree $\mathcal{T}$ of $(\mathcal{S}, \mathbf{t}, \mathcal{D})$. Then, the overall time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ will be $knc + a$, where a represents the time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ for the rules corresponding to the rest of the parts of the solution tree $\mathcal{T}$.

The proof starts by showing that when $n = 0$, the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to find all answer sets is equal to a and it monotonically increases if we increase n by 1 ($kc+a$ is clearly greater than a. Suppose that for all $i < n$, the theorem holds. Then, by induction, if we increase the number of methods applicable to task t to be $n + 1$, the overall time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate all answer sets will be $k(n + 1)c + a$ such that $k(n + 1)c + a > knc + a$.

Then, it follows that the total time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate all answer sets that correspond to $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ monotonically increases with the increasing number of subtasks of a particular task in $\mathcal{T}$. $\square$

Definition 22 (Range of a Variable: $\|v\|$)
The range of a variable v, denoted by range(v) is defined to be the set of all possible values defined for v. The cardinality of range(v) is denoted by $\|v\|$.

Definition 23 (The Universal set of atoms)
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the universal set of atoms, U, is defined as the set of all possible ground atoms that can ever be used to find all of the solutions for $(\mathcal{S}, \mathbf{t}, \mathcal{D})$.

We assume that for every atom a in U, the number of variables of a is a fixed number, say k. Furthermore, the range for every variable in a has the same length r.

Theorem 24 (Time Performance using $\mathfrak{Trans}(\cdot)$ (3))
Given a planning problem $(\mathcal{S}, \mathbf{t}, \mathcal{D})$, the corresponding logic program with negation $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$, and the solution tree $\mathcal{T}$, then, the time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate the set of all answer sets that correspond to $\mathbf{Sol}(\mathcal{S}, \mathbf{t}, \mathcal{D})$ *monotonically increases as k and/or r increase (see previous definition).*

Proof

The proof starts by defining the total number of ground instances, g, of a particular rule of $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$. Suppose that k and v, as described above, are fixed non-zero numbers initially. Let x be the number of atoms in that particular rule. Then, the total number of ground instances is $g = kvx$. Let $c(k, v)$ denote the time required by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ in order to make a rule ground, and fire it. If $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ fires N rules during the process of traversing the corresponding solution tree $\mathcal{T}$, then the total time required for $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ to generate all the possible answer sets is given by $Ng \times c(k, v)$. Then, by induction on k and/or v, we show that the theorem holds throughout the traversal process of $\mathcal{T}$ implemented by $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ in order to generate all of the answer sets corresponding to **Sol** ($\mathcal{S}$,$\mathbf{t}$,$\mathcal{D}$). □

As simple corollaries of the last theorem, we can conclude that the time for generating the set of all answer sets monotonically increases with (1) the number of preconditions of the methods, and (2) the number of atoms in the *add*- and *delete-lists* of the operators.

4.2 Experimental Study

In our experiments, we used two different planning domains:

The Travelling Domain: This domain is the one of the domains included in the distribution of SHOP planning system. The scenario for the domain as described in (Nau *et al.*, 1999) is that we want to travel from one location to another in a city. We have three locations: downtown, uptown, and park. There are three possible means of transportation: taxi, bus and foot. Taxi travel involves hailing the taxi, riding to the destination and paying the driver \$1.50 plus \$1.00 for each mile travelled. Bus travel involves hailing the bus, paying the driver \$1.00, and riding to the destination. Foot travel just involves walking., but the maximum feasible walking distance depends on the weather. Thus, different plans are possible depending on the weather conditions, the distance between our current location and the one we want to go, and how much money we have.

The Miconic-10 elevator Domain: This is the domain as described in Section 3.4. It is contained in a series of benchmarks `http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html` and it was recently used not only to measure the performance of various planners but also for other translation methods from planning problems into ASP (see `http://www.FCS.NMSU.Edu/~tson/asp_planner/`.

We describe our experiments in the following three subsections. We used the software package *smodels* v2.6—which is available at `http://www.tcs.hut.fi/Software/smodels/`—as the testing environment for our logic program encoding. We ran our experiments on a Solaris 2.6 Sun Ultra 1 machine. . However, we also tested our logic programs on the *DLV* system—which is available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.

4.2.1 Comparison with (Son et al.*, 2001)*

The section describes our comparison of the time performance of the logic programs produced by using our translation methodology with that of the logic-program encodings presented in (Son *et al.*, 2001). Note that the encoding methods proposed in (Son *et al.*, 2001) does not produce actual HTN encodings, rather they make use of only a few properties of HTN s—as they are introduced in (Erol *et al.*, 1994)—for implementing control knowledge in logic programs that perform action-based planning. In their paper, Son et al., showeed that employing that of control knowledge has increased the time performance of the logic program that encodes an action-based planner.

The problems that we used in these experiments are from `http://www.CS.NMSU.Edu/~tson/asp_planner`. Table 1 shows both our results and the results from (Son *et al.*, 2001), which were also obtained on the Smodels system. These experiments were run on an HP OmniBook 6000 Laptop with 128MB RAM and an Intel Pentium III 600 Mhz processor.

Problem	$\mathfrak{Trans}(\cdot)$	**(Son *et al.*, 2001)**
S1-0	0.150	0.100
S2-0	0.880	1.802
S3-0	6.300	22.682
S4-0	8.960	164.055
S5-0s1	2.530	57.952
S5-0s2	3.900	105.040
S6-0	53.340	no solution

Table 1. *Comparison of* HTN *Encoding with (Son* et al.*, 2001)*

The results clearly show that the logic programs produced by our translation methodology outperform the logic programs produced in (Son *et al.*, 2001). Our encoding was even able to solve a problem, for a solution could not be found by (Son *et al.*, 2001).

In this respect, these results confirm the fact that a *SHOP*-like HTN planning is an effective way for solving planning problems. They also illustrate that our translation method provides a way to produce efficient HTN-logic programs with ASP semantics to solve planning problems compared to other action-based encoding methodologies that use some HTN concepts as domain control knowledge. We believe that this is due to the fact that HTN-planning is more expressive than action-based planning (**?**). Thus, all of the planning problems can be represented in HTN formalism. For this reason, our translation methodology offers an efficient way solving planning problems by using logic programs with answer set semantics.

4.2.2 The Effect of Grounding

We hypothesize that our translation methodology provides more efficient logic programs with ASP semantics if the system on which those programs are implemented allows the usage of free variables in the programs. Otherwise, the system tries to make every rule ground in the input program, which is not an appropriate behaviour in planning. Most of the recent planning systems—such as *SHOP* (Nau *et al.*, 1999) , *TALPlanner* (**?**), etc.—can work on planning-problem descriptions with free variables and these systems are proven to be faster then those which reequire grounding.

As we described earlier, the *smodels* system cannot work on the logic programs with free variables. To test our hypothesis, we applied our translation methodology to our elevator and travelling examples to produce logic programs on a different system called *DLV*. *DLV* is a deductive database system, and can be used as a logic programming system as well. It implements stable model semantics and it supports the usage of free variables in the input logic programs.

Tables 2 show our results on the travelling problems. As it can be seen, our programs are much more faster on *DLV*, then on *smodels*. This is because of the fact that, as described in Section 3.4, *smodels* work on ground logic programs. Because of that we have to define type predicates for each variable in the problem domain as well as all possible ground instances of the atoms that can ever be used in the planning process. The result is that as the number of variables and the number of their possible instantiations increase, the time performance of the logic program decreases (see Theorem 24). However, we do not have such constraints on *DLV* since it can work on programs with free variables. Thus, the effect of grounding can be clearly seen in our results on travelling domain.

On the elevator problems, however, the performances of our programs are almost the same (see Table 3). The reason for this is the fact that the elevator domain, by its definition, enforces the input programs to be more ground than the travelling domain. Thus, no matter they are implemented on either *smodels* or *DLV*, the programs need to be ground. Therefore, in this domain the effect of grounding cannot be seen clearly.

4.2.3 Comparison with SHOP

Encouraged by the performances of the logic programs produced by our translation, we prepared an experiment set in which we compared the time performances of our logic-program encodings on the travelling examples with those of the SHOP planning system itself on the same domain.

For this experiment set we designed three independent variables, namely the destination location, the weather condition and the financial status of the traveller. We assumed that the traveller always starts travelling from the downtown of the city. The treatments for these independent variables are: the destination location can be uptown or park, the weather can be good or bad, and the traveller can be

Problem	*smodels*	*DLV*
P1	3.23	0.2
P2	2.23	0.12
P3	2.19	0.22
P4	2.08	0.10
P5	2.2	0.19
P6	2.18	0.11
P7	2.21	0.19
P8	2.15	0.08

Table 2. *Comparison of* smodels *and DLV using* $\mathfrak{Trans}(\cdot)$ *(1)*

Problem	*smodels*	*DLV*
S1-0	0.150	0.51
S2-0	0.880	1.30
S3-0	6.300	6.61
S4-0	8.960	6.66
S5-0s1	2.530	4.06
S5-0s2	3.900	3.76
S6-0	53.340	54.54

Table 3. *Comparison of* smodels *and DLV using* $\mathfrak{Trans}(\cdot)$ *(2)*

either rich (i.e., have sufficient money for travelling with taxi) or broke (i.e, has no money at all).

As it can be seen from the results of our experiment (see Table 4), the performance of our logic programs are comparable to that *SHOP*. Given the fact that *SHOP* is proven to be one of the fastest and efficient planners in the AIPS-2000 planning competition (Bacchus, 2001), these results suggest that our translation methodology

Problem	*SHOP*	$\mathfrak{Trans}(\cdot)$ on *DLV*
P1	0.026	0.20
P2	0.002	0.12
P3	0.003	0.22
P4	0.002	0.10
P5	0.004	0.19
P6	0.009	0.11
P7	0.003	0.19
P8	0.003	0.08

Table 4. *Comparison of* $\mathfrak{Trans}(\cdot)$ *with SHOP (no Grounding)*

introduces a way of providing very efficient solutions to planning problems using logic programming with ASP semantics and it has the potential to be the most competitive approach in the logic-programming literature with the actual planning systems.

In the near future, we will test our system on more planning domains and compare our approach with other well-known planning systems. We are also planning to implement our approach on two systems, namely the *XSB* system ((Rao *et al.*, 1997)) and the front-end software developed by P. Bonatti for *smodels* ((Bonatti, 2001b; Bonatti, 2001a)), both of which can handle free variables like the *DLV* system.

4.2.4 Experimental Verification of Our Theorems

This section describes four sets of experiments with respect to Theorems 20, and 21. We introduced two independent variables, one for each experiment: the number of applicable methods for a particular task $||m||_t$, and the number of subtasks of a particular task, $||t||_t$. In each experiment, we measured the time performance of our logic program encoding produced by our translation methodology on the travelling problems.

We designed different number of treatments for each of our independent variable. For the independent variables about a particular task, we chose the travelling task (i.e., the task $travelXY$) in *SHOP* notation). The treatments for our independent variables are shown in Table 5.

The results are shown in Table 6 and Table 7. These results corroborate with our theoretical results in Theorem 20 and 21, respectively. As it can be seen from these

Independent Variable	Treatments
$\|\|m\|\|_t$	1, 2, 3, 4
$\|\|t\|\|_t$	2, 3, 4

Table 5. *Treatments of Independent Variables*

tables, the time required to generate all the answer sets in **Sol** ($\mathcal{S}$,**t**,$\mathcal{D}$) increases with the number of methods applicable to a particular task and with the number of subtasks of a particular task.

Independent Variable	1. Treat.	2. Treat.	3. Treat.	4. Treat.
$\|\|m\|\|_t$	6.1	17.68	29.59	41.77

Table 6. *Performance of* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *wrt. Number of Methods.*

Independent Variable	1. Treat.	2. Treat.	3. Treat.
$\|\|t\|\|_t$	13.75	17.68	39.33

Table 7. *Performance of* $\mathfrak{Trans}((\mathcal{S}, \mathbf{t}, \mathcal{D}))$ *wrt. Number of Subtasks.*

5 Conclusions and Future Research Directions

In this paper, we described a way to encode HTN-planning problems into logic programs under the answer set semantics. This transformation is not only sound and complete, but it also corresponds closely to HTN-planning systems which generate plans by using ordered task decompositions. Previous encodings (as first introduced in (Dimopoulos *et al.*, 1997)) do consider action-based planning or they take a special view of HTN planning (as constraint-based planning, like in (Son *et al.*, 2001)).

To test our approach, we used it to create both *smodels* logic programs and *DLV* logic programs, for two different AI planning domains: the Travelling Domain, and the “first track” version of the Miconic 10 Elevator Planning Domain. Here is a summary of our experimental results and what we believe they signify:

- In our experiments on the Miconic 10 domain, our *smodels* logic programs clearly outperformed the corresponding ones described in (Son *et al.*, 2001), which are based on answer set semantics. This, we believe, is due largely to the HTN-style control knowledge that our translation methodology encodes into the logic programs.

- Although our logic-program encodings on *smodels* outperformed those of (Son *et al.*, 2001), they were not competitive with *SHOP*, which is a state-of-the-art AI planning system. We believe one of the reasons for this is that *smodels* require grounding, which creates combinatorially many ground instances of the clauses in the logic program. For any given problem instance, most of these clauses are likely to be irrelevant.
- Our overall translation methodology does not rely on grounding. Grounding is merely used here because many available systems, notably *smodels*, require it. *DLV*, on the other hand, allows for free variables, but does not allow function symbols, which come in handy in *smodels*. We have included in this paper our first experiments in applying our methodology for programs with free variables. In our experiments on the Travelling Domain using our method together with *DLV*, we got a speed-up of two orders of magnitude compared to *smodels*. However, the performance was still about 1.5 orders of magnitude worse than *SHOP*, one of the best planning systems on the market.

We emphasize the fact that our method does not use any particular features of the engine for computing answer sets. Obviously, taking advantage of the particular search method of *smodels*, or the bottom-up evaluation of *DLV*, it would be possible to write even more efficient translations. But our aim is to develop a translation that is independent of the underlying nonmonotonic engine.

As a byproduct, we believe our method can be easily used as transferring benchmarks from the planning community to benchmarks for comparing nonmonotonic systems based on computing answer sets. This is because our method is very general and does not rely on the features of a particular system. Due to lack of time, we were not yet able to test the benchmarks on the *XSB* system, a Prolog system which not only allows function symbols but also free variables at the same time. These are features that neither *smodels* nor *DLV* provide. We believe that we can get a competitive planning system once we can apply our translation into a nonmonotonic system with these two features.

We are also planning to compare our method with *smodels* equipped with a front-end to allow for (restricted use of) free variables ((Bonatti, 2001b; Bonatti, 2001a)). The latter system has been developed by Piero Bonatti and is a front-end system that can be added to any system computing answer sets and based on grounding. This would also allow for comparisons of systems with built-in grounding to those who do not require this (but are, in general, slower). Again, we believe that serious benchmarks from the planning community can help a lot to evaluate nonmonotonic systems.

Our overall aim is to investigate to what extent state-of-the-art nonmonotonic theorem provers can compete with dedicated planners (in particular those based on HTN) and what lessons we can learn from the different translation methods. We expect that optimal translations (if they exist) depend on the particular application area. Developing a methodology to determine or classify such domains seems to us to be worthwhile.

Acknowledgments

This work was supported in part by the following grants, contracts, and awards: Air Force Research Laboratory F306029910013 and F30602-00-2-0505, Army Research Laboratory DAAL0197K0135, and the University of Maryland General Research Board. Opinions expressed in this paper are those of authors and do not necessarily reflect opinion of the funders.

References

Apt, K. R., Marek, V., Truszczynski, M., & Warren, D. S. (eds). (1999). *The Logic Programming Paradigm: Current Trends and Future Directions*. Berlin: Springer.

Bacchus, F. (2001). Aips'00 planning competition. *Ai magazine*, **22**(3).

Baral, C., & Tuan., L. (2001). Reasoning about actions in a probabilistic setting. *In common sense*.

Bonatti, P.A. (2001a). Prototypes for reasoning with infinite stable models and function symbols. *Pages 416–419 of:* Eiter, Th., Truszczyński, M., & Faber, W. (eds), *Logic programming and non-monotonic reasoning, proceedings of the sixth international conference*. LNCS 2173. Berlin: Springer.

Bonatti, P.A. (2001b). Reasoning with infinite stable models. *Pages 603–608 of: Proceedings of ijcai-01*.

Chen, Weidong, & Warren, David S. (1996). Tabled Evaluation with Delaying for General Logic Programs. *Journal of the acm*, **43**(1), 20–74.

Dimopoulos, Yannis, Nebel, Bernhard, & Koehler, Jana. (1997). Encoding planning problems in nonmonotonic logic programs. *Pages 169–181 of: Proceedings of the Fourth European Conference on Planning, ECP*.

Dix, Jürgen, Furbach, Ulrich, & Niemelä, Ilkka. (2001). Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. *Pages 1121–1234 of:* Voronkov, Andrei, & Robinson, Alan (eds), *Handbook of automated reasoning*. Elsevier-Science-Press.

Eiter, Thomas, Leone, Nicola, Mateis, Cristinel, Pfeifer, Gerald, & Scarcello, Francesco. (1998). The KR System `dlv`: Progress Report, Comparisons and Benchmarks. *Pages 406–417 of: Proceedings sixth international conference on principles of knowledge representation and reasoning (kr'98)*.

Erol, K., Hendler, J., & Nau, D.S. (1994). Umcp: A sound and complete procedure for hierarchical task-network planning. *Proceedings of aips-94*.

Gelfond, M., & Lifschitz., V. (1998). Action languages. *Electronic transactions on ai*, **3**(16).

Giunchiglia, E., & Lifschitz, V. (1998). An action language based on causal explanation: preliminary report. *Pages 623–630 of: Proc. aaai-98*.

Lifschitz, V. (1999). Action languages, answer sets and planning. *Pages 357–373 of: The logic programming paradigm: a 25-year perspective, springer-verlag*.

McCain, N., & Turner, H. (1997). Causal theories of action and change. *Pages 460–465 of: Proceedings of the 14th national conference on artificial intelligence (aaai-97)*. Menlo Park, CA: AAAI Press.

Nau, D.S., Cao, Y., Lotem, A., & Muñoz-Avila, H. (1999). Shop: Simple hierarchical ordered planner. *Proceedings of ijcai-99*.

Nau, D.S., Cao, Y., Lotem, A., & Muñoz-Avila, H. (2000). *SHOP and M-SHOP: Planning with Ordered Task Decomposition.* Tech. rept. CS TR 4157. University of Maryland. Submitted for publication.

Nau, D.S., Cao, Y., Lotem, A., Muñoz-Avila, H., & Mitchell, S. (2001). Total-order planning with partially ordered subtasks. *Proceedings of ijcai-01.*

Niemelä, Ilkka, & Simons, Patrik. (1996). Efficient Implementation of the Well-founded and Stable Model Semantics. *Pages 289–303 of:* Maher, M. (ed), *Proceedings of the joint international conference and symposium on logic programming.* Bonn, Germany: The MIT Press.

Pearl, Judea. (1988). *Probabilistic reasoning in intelligent systems.* San Mateo: Morgan Kaufmann.

Rao, Prasad, Sagonas, K., Swift, T., Warren, D. S., & Freire, J. (1997). XSB: A System for Efficiently Computing Well-Founded Semantics. *Pages 430–440 of:* Dix, J., Furbach, U., & Nerode, A. (eds), *Logic programming and non-monotonic reasoning, proceedings of the fourth international conference.* LNAI 1265. Berlin: Springer.

Sacerdoti, E. (1977). *A structure for plans and behavior.* American Elsevier Publishing.

Son, T.C., Baral, C., & McIlraith., S. (2001). Planning with domain-dependent knowledge of different kinds – an answer set programming approach. Eiter, Th., Truszczyński, M., & Faber, W. (eds), *Logic programming and non-monotonic reasoning, proceedings of the sixth international conference.* LNCS 2173. Berlin: Springer.

Turner, H. (1997). Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *The journal of logic programming*, **31**(1-3), 245–298.

Wilkins, D.E. (1988). *Practical planning - extending the classical ai planning paradigm.* Morgan Kaufmann.